Higher-order strictness analysis over non-flat domains

C. Ernoult

INRIA

UNITE DE RECHERCHE
INRIA-ROCQUENCOURT

Institut National
de Recherche
en Informatique
et en Automatique

Domaine de Voluceau
Rocquencourt
B.P.105
78153 Le Chesnay Cedex
France
Tél. (1) 39 63 55 11

Rapports de Recherche

N° 1190

Programme 1
Programmation, Calcul Symbolique
et Intelligence Artificielle

HIGHER-ORDER STRICTNESS ANALYSIS OVER NON-FLAT DOMAINS

Christine ERNOULT

Mars 1990

RR-1190

Higher-Order Strictness Analysis over non-flat domains [0]

Christine Ernoult

Yale University
Department of Computer Science
P.O. Box 2158, Yale Station
New Haven, CT 06520, U.S.A.
ernoult@cs.yale.edu

Abstract

Strictness analysis has been investigated in order to cover in one hand higher-order strictness analysis, on the other hand strictness analysis on non-flat domains. This paper presents a general method extending higher-order strictness analysis in order to subsume non-flat domains. The method uses the notion of strictness pairs and the notion of projection from domain theory. It allows to deal with the "n-th" strictness of a component of any kind of data structures.

Analyse de nécessité d'ordre supérieur pour des domaines non-plats

Christine Ernoult

Yale University
Department of Computer Science
P.O. Box 2158, Yale Station
New Haven, CT 06520, U.S.A.
ernoult@cs.yale.edu

Résumé

Diverses études sur l'analyse de nécessité ont été menées concernant, soit l'analyse d'ordre supérieur, soit l'analyse de nécessité pour des domaines non-plats. Ce rapport présente une méthode générale qui étend l'analyse de nécessité d'ordre supérieur afin de pouvoir intégrer les domaines non-plats. Cette méthode utilise la notion de paires de nécessité et la notion de projection de la théorie des domaines. Elle permet de déterminer la "$n^{\text{ième}}$" nécessité d'une composante d'une structure de donnée quelconque.

[0] This work was supported by INRIA (Institut National de Recherche en Informatique et Automatique), France.

1 Introduction

Strictness analysis is a compile-time method which can determine which arguments a function is sure to evaluate. More formally, a function $f(x_1, ..., x_i, ..., x_n)$ is strict in x_i if $f(x_1, ..., x_{i-1}, \perp, x_{i+1}, ..., x_n) = \perp$ for all values of x_j, $j \neq i$. Strictness analysis is widespread used in compiler optimization, program transformation, and more recently parallel evaluation of functional programming languages [9], [2].

One of the most promising techniques of strictness analysis is by abstract interpretation [4], [8]. Mycroft's pioneering work on strictness analysis was confined to first-order functions over flat domain of basic data. This was extended in one hand to include higher-order functions [3] , [5] and polymorphism [1], on the other hand to cover non-flat domains [13], [14].

Our interest has been trying to reconcile the differences between higher-order strictness analysis and strictness on non-flat domains. More precisely, one can make the observation that strictness on non-flat domains amounts to special treatments of data structures such as lists; but in fact lists can be represented as higher-order functions. So, the natural question to ask is whether conventional higher-order strictness analysis can be extended to subsume non-flat domains. This would certainly be a more general solution, and may make extensions to other kind of data structures much easier.

After studying the higher-order strictness analysis of [5], still limited to flat domains, and the strictness analysis on non-flat domains of [14], still limited to first-order languages, we propose a method to deal with higher-order strictness analysis over non-flat domains.

This method is based on forwards analysis and on the notion of projection from domain theory [11] and, so, built on existing mathematical foundations and intuitions. This method extends the work of [5] based on strictness pairs in order to deal with higher-order functions and uses the notion of finite domain of projection from [14] in order to formalize, for instance some kinds of strictness of interests for lazy lists, head strictness alone, tail strictness alone, and of course both head and tail strictness. This method is general enough to deal with other kinds of data structures and to find strictness of a component of a structure. It will be used as a basic framework to two further extensions : 1) strictness analysis in the context of an entire program (what allows to find informations such as head strictness in a head strict context) and 2) polymorphic strictness analysis.

This paper is organized as follows. Section 2 gives some preliminary notions of higher-order strictness analysis and projections. A simple approach to deal with higher-order strictness analysis over non-flat domains is given in section 3 through the example of finite domain of projections over lists allowing to find head strictness and/or tail strictness. Section 4 extends the method to deal with the "n-th" strictness analysis of other kind of data structures and lift restrictions that we have made in order to simplify the exposition of our approach. In section 5 we conclude.

Notation: We use fairly traditional notation from denotational semantics. Domains are treated as "pointed cpos" – chain-complete partial orders with a unique least element called "bottom," which for a domain D is written $\perp_D$ or just $\perp$ when the domain is clear from context. $A* \rightarrow B$ denotes the domain $B+(A \rightarrow B)+(A \rightarrow A \rightarrow B)+\cdots$. We write "$d \in D = ...$" to define the domain (or set) D with "typical" element d. All domain/subdomain coercions are omitted when clear from context. It is convenient to write n-ary functions as "$\lambda x_1\ x_2\ ...\ x_n.\ exp$," and when $n = 0$ we interpret this to mean just exp. Double brackets are used to surround syntactic objects, as in $\mathcal{E}[\![exp]\!]$; square brackets are used for environment update, as in $env[e/x]$, which is the same as $\lambda id.$ if $id = [\![x]\!]$ then e else $env(id)$; and angle brackets are used for tupling, as in $\langle e_1, e_2, e_3 \rangle$. The notation $env[e_i/x_i]$ is shorthand for $env[e_1/x_1,\ ...,\ e_n/x_n]$, where the subscript bounds are inferred from context. Thus "new" environments are created by $\perp[e_i/x_i]$.

2 Preliminaries

In this section we sketch some preliminary notions about higher-order strictness based on strictness pairs and projections for strictness analysis. These notions will be useful to present our method.

2.1 Standard Higher-Order Semantics

We give some preliminary notion of syntax and standard semantics for a higher-order language with the full power of the untyped lambda calculus with constants. We consider all functions to be "curried". We allow nested groups of equations, and we take the value of the right-hand side of the first equation as the value of the equation group.

Abstract Syntax

$$
\begin{array}{rcll}
c & \in & Con & \text{constants} \\
x, f & \in & V & \text{variables, either bound variables or function names} \\
eg & \in & EqGr & \text{equations groups, where } eg ::= \{f_1 = e_1, ..., f_n = e_n\} \\
e & \in & Exp & \text{expressions, where } e ::= c \mid x \mid f \mid e_1 \rightarrow e_2, e_3 \mid (\lambda x.e) \mid e_1\ e_2 \mid eg \\
pr & \in & Prog & \text{programs, where } pr ::= eg
\end{array}
$$

Standard Semantic Domains

$$
\begin{array}{rcll}
Bas & = & Int + Bool + \{Nil\} & \text{domain of basic values} \\
D & = & Bas + (D \rightarrow D) & \text{domain of denotable values} \\
Env & = & V \rightarrow D & \text{domain of environments}
\end{array}
$$

Standard Semantic Functions

$$
\begin{array}{rcll}
\mathcal{E}: & Exp \rightarrow Env \rightarrow D & & \\
\mathcal{P}: & Prog \rightarrow D & & \\
\mathcal{K}: & Con \rightarrow D & & \text{assumed given}
\end{array}
$$

$$
\begin{array}{rcl}
\mathcal{E}[\![c]\!]env & = & \mathcal{K}[\![c]\!] \\
\mathcal{E}[\![x]\!]env & = & env[\![x]\!] \\
\mathcal{E}[\![e_1 \rightarrow e_2, e_3]\!]env & = & \mathcal{E}[\![e_1]\!]env \rightarrow \mathcal{E}[\![e_2]\!]env, \mathcal{E}[\![e_3]\!]env \\
\mathcal{E}[\![\lambda x.e]\!]env & = & \lambda v.\ \mathcal{E}[\![e]\!]env[v/x] \\
\mathcal{E}[\![e_1\ e_2]\!]env & = & (\mathcal{E}[\![e_1]\!]env)(\mathcal{E}[\![e_2]\!]env) \\
\mathcal{E}[\![\{f_1 = e_1, ..., f_n = e_n\}]\!] & = & \mathcal{E}[\![e_1]\!]env' \\
 & & \text{whererec } env' = env[\mathcal{E}[\![e_1]\!]/f_1, ..., \mathcal{E}[\![e_n]\!]/f_n] \\
\mathcal{P}[\![pr]\!] & = & \mathcal{E}[\![pr]\!]null - env
\end{array}
$$

2.2 Higher-Order Strictness Analysis on Flat Domains

Let us define a higher-order strictness analysis (taken from [5]). This analysis has considered functions passed as parameters in function calls or returned as values from expressions (including function calls). For example, consider the simple program in the notation of Haskell [7]

```
f x   = x
g x   = x+1
h a b = (if (a=0) then f else g) b
```

This analysis is able to detect the fact "h is strict in a and in b".

The important observation to be made is that an expression has not only a "direct strictness" (the set of variables which are evaluated when it is), but also a "delayed strictness" (the set of variables which are evaluated when the expression is applied).This suggests that the strictness property should perhaps be captured by an object, the domain of stricness pairs Sp defined by:

$$Sp = Sv \times (Sp \rightarrow Sp)$$

With every expression exp in a strictness environment $senv$, we associate a strictness pair that provides properties of exp both as an isolated value and as a function to be applied.

$$\mathcal{S}[\![exp]\!]senv = \langle sv, sf \rangle$$

The set of strict variables is a flat domain and so this analysis is unable to get strictness informations for structured data types such as tail strictness for lists.

In this analysis, a strictness pair for $cons$ in the definition of $\mathcal{K}_s$ is not provided. It turns out that one can define $cons$, car, cdr as higher-order functions in the pure lambda calculus.

We recall now the non-standard semantics of this analysis.

Non-Standard Semantic Domains

V,	variables of interest
$Sv = \mathcal{P}(V)$,	the powerset of V
$Sp = Sv \times (Sp \rightarrow Sp)$,	domain of strictness pairs
$Senv = V \rightarrow Sp$,	the strictness environments

For simplicity we use the following subscript notation on strictness pairs: $\langle sv, sf \rangle_v = sv$ and $\langle sv, sf \rangle_f = sf$. We also define a special error element $serr = \lambda\hat{x}.\ \langle \emptyset, serr \rangle$ to be used when expression has been applied "too many times". We write $e_1 \sqcap e_2$ to mean $\lambda\hat{x}.\ \langle (e_1\ \hat{x})_v\ \cap\ (e_2\ \hat{x})_v, (e_1\ \hat{x})_f\ \sqcap\ (e_2\ \hat{x})_f \rangle$.

Non-Standard Semantic Functions

$\mathcal{K}_s$:	$Con \rightarrow Sp$,	maps constants to Sp (assumed given)
$\mathcal{S}$:	$Exp \rightarrow Senv \rightarrow Sp$,	maps expressions to strictness pairs
$\mathcal{P}_s$:	$Prog \rightarrow Senv$,	gives meaning to programs

$$
\begin{array}{rcl}
\mathcal{S}[\![k]\!]senv & = & \mathcal{K}_s[\![k]\!] \\
\mathcal{S}[\![x]\!]senv & = & senv[\![x]\!] \\
\mathcal{S}[\![e_1 \rightarrow e_2, e_3]\!]senv & = & \langle (\mathcal{S}[\![e_1]\!]senv)_v \cup ((\mathcal{S}[\![e_2]\!]senv)_v \cap (\mathcal{S}[\![e_3]\!]senv)_v), (\mathcal{S}[\![e_2]\!]senv)_f \sqcap (\mathcal{S}[\![e_3]\!]senv)_f \rangle \\
\mathcal{S}[\![\lambda x.e]\!]senv & = & \langle \emptyset, \lambda\hat{x}.\ \mathcal{S}[\![e]\!]senv[\hat{x}/x] \rangle \\
\mathcal{S}[\![e_1\ e_2]\!]senv & = & \langle (\mathcal{S}[\![e_1]\!]senv)_v \cup sv, sf \rangle \\
 & & \text{where } \langle sv, sf \rangle = (\mathcal{S}[\![e_1]\!]senv)_f\ (\mathcal{S}[\![e_2]\!]senv) \\
\mathcal{S}[\![\{f_1 = e_1, ..., f_n = e_n\}]\!]senv & = & \mathcal{S}[\![e_1]\!]senv' \\
 & & \text{whererec } senv' = senv[\mathcal{S}[\![e_1]\!]senv'/f_1, ..., \mathcal{S}[\![e_n]\!]senv'/f_n] \\
\mathcal{P}_s[\![\{f_1 = e_1, ..., f_n = e_n\}]\!]senv & = & senv' \\
 & & \text{whererec } senv' = senv[\mathcal{S}[\![e_1]\!]senv'/f_1, ..., \mathcal{S}[\![e_n]\!]senv'/f_n]
\end{array}
$$

Thus the abstract "meaning" of a program is a "strictness environment" that binds the top level functions to strictness pairs. In shorthand, we write $\hat{f}$ to refer to the strictness pair of a function f.

Computing the Least Fixpoint of Strictness Pairs

Given this higher-order analysis, how does one compute the fixpoint of the resulting mutually recursive equations defining strictness pairs? The domain $Sp = Sv \times (Sp \rightarrow Sp)$ seems to have unbounded "depth", making the computation seemingly difficult. Fortunately, the second element in a pair almost always degenerates to "serr" at some point, which can be represented compactly. Further, strictness pairs are almost never applied more than a finite number of times. Thus one may use the standard technique of starting with an initial approximation $\perp_{Sp}$ for all strictness pairs, and iterating in the normal way to refine the approximations to whatever degree is necessary for the particular application.

In general it is not guaranteed to terminate! The reason is that there is occasion when a strictness pair needs to be applied an infinite number of times. This aspect of untyped lambda calculus is an unfortunate one, considering that most programs have little need for such functions. The only solution to this problem currently is to impose a weak type discipline that disallocates functions whose type is of arbitrary "order" or "depth". In particular, most versions of typed lambda calculus provide the necessary constraints.

This analysis so far based on abstract interpretation is unable to perform strictness analysis for data types over non-flat domains such as lazy lists.

2.3 Projections for Strictness Analysis

Strictness analysis on non-flat domains has received a great deal of attention. An early proposal in this direction was made by Hughes based on analysis of the context in which an expression may be evaluated. Wadler discovered an approach on non-flat domains using abstract interpretation over finite domains. Hughes and Wadler's work provided a new description of contexts which are identified with the notion of projections from domain theory. Moreover, finite domains are given for contexts over lists.

This analysis is able to get informations such as head-strictness, tail-strictness for lazy lists. But, the method is still limited to first-order, monomorphic languages. In this section we sketch the ideas in [14], showing how projections can be used for strictness analysis, in particular analysis of functions on lazy data structures.

First, we introduce head-strictness and tail-strictness and formalize these with projections.

Head and Tail Strictness

Let us first give three functions on lazy lists to serve as examples.

```
length [ ]   = 0
length(x:xs) = 1+length xs

before [ ]   = [ ]
before(x:xs) = [ ],              x = 0
             = x : before xs ,   otherwise

sum[ ]    = 0
sum(x:xs) = x + sum xs
```

Let us first consider the function *length*. It is strict, that is $length\ \bot = \bot$, but we can say more than this. *Length* evaluates the entire spine of its arguments, i.e. all the tails. We can express denotationally

$$length(x_1 : x_2 : ... : x_n : \bot) \ = \ length\ \bot \ = \ \bot \ for\ any\ x_i$$

If we want to pass *length* a list ending in $\bot$, we might as well pass $\bot$ instead. We can define a function T which removes useless information from an argument of length.

$$\begin{array}{lcl} T(x_1 : ... : x_n) & = & (x_1 : ... : x_n) \\ T(x_1 : ... : x_n : \bot) & = & \bot \end{array}$$

We can express our observation about length in the form $length = length\ o\ T$ and we will say f is tail-strict if $f = f\ o\ T$. T can be thought of as evaluating the spine of a lazy list, and the equation says that evaluating the spine of f's argument before calling it cannot change its result.

It is easy to see that

$$\begin{array}{lcl} T\ o\ T & = & T \\ T & <= & ID \end{array}$$

and so T is an example of a domain-theoric projection [12].

Now consider the function *before*. Since the first thing *before* does is to check whether the first element is zero or not. It returns $\bot$ if that element is undefined and it does not distinguish between a list with an undefined element and a list which becomes undefined at the same point.

$$\begin{array}{lcl} before(\bot : xs) & = & \bot \\ before(1 : \bot : 0) & = & (1 : \bot) \\ before(1 : 2 : 0 : \bot : 3) & = & (1 : 2) \end{array}$$

We can also define a projection H such that

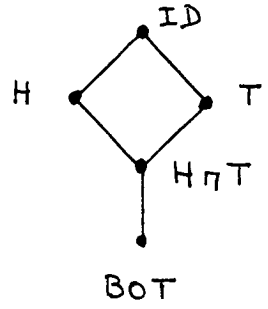
$$\begin{array}{lcll} H(x_1 : ... : x_n : \bot : ...) & = & (x_1 : ... : x_n : \bot) & if\ each\ x_i \neq \bot \\ H(x_1 : ... : x_n) & = & (x_1 : ... : x_n) & \end{array}$$

We will say that f is head-strict if $f \circ H = f$.

The function *sum* is both head-and-tail strict, it returns $\perp$ if any part of its argument is undefined. We will write H $\sqcap$ T the corresponding projection.

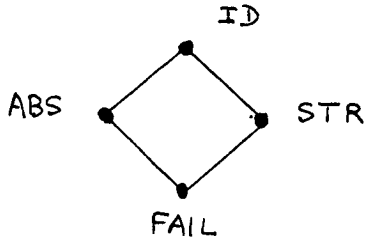
Projections

Projections form a complete lattice under the $\sqsubseteq$ ordering with ID at the top and BOT at the bottom, where BOT is the function defined by BOT u=$\perp$ for all u. For example, the projections we have seen form the following lattice.



We have used projections to characterize concepts such as head-strict and tail-strict. We can also characterize ordinary strictness defined by $f\perp = \perp$, using projections but it requires some extensions. In particular, we must use projections to say that is necessary that a value be more defined than $\perp$. The domains are extended by adding a new bottom element, $\natural$ beneath the existing bottom $\perp$, so $\natural \sqsubset \perp \sqsubseteq$ u, for every u$\in D$. The interpretation of $\alpha\, u = \natural$ will be that α requires a value more defined than u. To define strictness a new projection STR is used and does not accept $\perp$ so we must have $STR\ \perp = \natural$. ID is still the top of element of the domain of projections, BOT is no longer the bottom and is rechristened ABS. The new bottom is FAIL defined by $FAIL\ u = \natural$, for all u.

The four projections have the following ordering.



This is just the subdomain of the domain of projections over D. In labelling a subexpression e with one of these projections, we get the following interpretation:

- FAIL : no value that could be returned by e is acceptable
- ABS : the value of e is ignored
- STR : the value of e is required
- ID : the value of e may be required or ignored.

We will call $\alpha : D_\natural \rightarrow D_\natural$ a projection over D. The strict part of a projection α is $\alpha' = \alpha \sqcap$ STR (for example ID' = STR, ABS' = FAIL). α is strict if it is equal to its strict part or equivalently if $\alpha \sqsubseteq$ STR. This implies that α is called strict iff $\alpha\ \perp = \natural$. The problem of analysing strictness in α can be reduced to analysing strictness in α'. Notice that $\alpha = \alpha' \sqcup$ ABS.

Let us define an operation & such that: if γ_1 and γ_2 are some projections, we have

$$\begin{array}{lll} (\gamma_1 \ \& \ \gamma_2)\ u & = \ \natural, & \textit{if } \gamma_1\ u = \natural \textit{ or } \gamma_2\ u = \natural \\ & = \ \gamma_1\ u \sqcup \gamma_2\ u, & \textit{otherwise} \end{array}$$

$\forall u \in D,\ \natural \sqsubset \bot \sqsubseteq u$ (*recall* $FAIL\ u = \natural$)

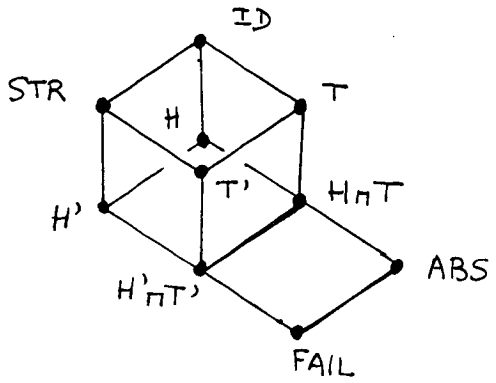
$$FAIL\ \&\ \gamma = FAIL \textit{ and } ABS\ \&\ \gamma = \gamma$$

This operation satisfies many laws: it is commutative, associative, idempotent, has ABS as a unit, FAIL as a zero, distributes over $\sqcup$.

Finite domains of projections

Let D_C be the finite domain of projections over D. For instance, if D is INT then D_C might be {STR,ID}. Let $LIST\ D$ be the non-flat domain of lists whose elements are in domain D, for example LIST INT. We have already discussed two projections over $LIST\ D$, namely H and T. The finite domain $LIST_C\ D_C$ of projections over $LIST\ D$ consists of the eight projections STR, H', T', H' $\sqcap$ T', ID, H, T, H $\sqcap$ T, plus ABS and FAIL.

The projections H and T were defined before $\natural$ was introduced and so H$\bot$ = $\bot$ and T$\bot$ = $\bot$, therefore H and T are non strict. The corresponding versions are H' and T' with H'=H $\sqcap$ STR and T' = T $\sqcap$ STR. A diagram of this domain is given.



3 A Simple Approach

Our approach is based on the observation that strictness analysis on non-flat domains amounts special treatments of data structures such as lists (for example, head strictness); but lists can be represented as higher-order functions. In the analysis of [5], all the functions are curried. So *cons*, *head* and *tail* can be represented by:

$$\begin{array}{lll} (x : xs) & : & \lambda c.\ c\ x\ xs \\ hd\ a & : & a\ (\lambda h.\lambda t.\ h) \\ tl\ a & : & a\ (\lambda h.\lambda t.\ t) \end{array}$$

We can also define some primitive functions related to the lists.

$$\begin{array}{lll} nil & = & \lambda s.\ false \\ null?\ xs & = & (xs\ \emptyset\ ) = false \rightarrow true,\ false \end{array}$$

Thus, the "case" function is rewritten as:

$$case\ lst\ e_1\ e_2 \ = \ null?\ lst \rightarrow e_1, e_2$$

As an example, $(x_1 : x_2 : x_3)$ is represented by $\lambda s_1.\ s_1\ x_1(\lambda s_2.\ s_2\ x_2\ x_3)$.

Let us consider a simple example such as *before* which is head-strict. This example can be rewritten in a curried form:

$$before \ = \ \lambda xs.null?\ xs \rightarrow Nil,\ ((xs\ (\lambda h.\lambda t.\ h)\ =\ 0) \rightarrow \lambda c.\ c(xs\ (\lambda h.\lambda t.\ h))(before\ xs\ (\lambda h.\lambda t.\ t)))$$

Let us define $hh = \lambda h.\lambda t.\ h$, $tt = \lambda h.\lambda t.\ t$ and $g = xs$. We can rewrite the above expression in

$$before = \lambda g.null?\ g \rightarrow Nil,\ ((g\ hh) = 0) \rightarrow \lambda c.\ c(g\ hh)(before\ (g\ tt)))$$

Before is head strict because the first thing it does is to check wether is the first element is equal to 0. We can capture this strictness information by combining the projection H with the strictness information of the first element of the curried-form list.

Recall that H means not head-strict and its strict version is H', where H'= H $\sqcap$ STR and H = H' $\sqcup$ ABS. The list g can be rewritten as $\lambda c.\ c\ h\ t$.

Recall also that we are extending the higher-order strictness analysis of [5]. So the domain of strictness pairs has to be modified in order to include other informations than strict or non-strict. In the case of lists, we desire more informations such as head-strict or tail-strict. The finite domain $LIST_C\ D_C$ of projections over LIST D (presented in the previous section) contains such informations. So we get the new domain of strictness pairs

$$SP = LIST_C\ D_C \times (SP \rightarrow SP)$$

In order to capture the behaviour of the above example, we should associate to the λ-function hh, an object giving the abstract meaning that we desire.

$$\mathcal{S}[\![\lambda h.\lambda t.\ h]\!]senv = \langle ABS, \lambda\hat{h}.\langle H \sqcap \hat{h}_v, \lambda\hat{t}.\langle ABS, serr\rangle\rangle\rangle$$

where ABS is the projection meaning that the value is ignored. In the same way, we can define for tt

$$\mathcal{S}[\![\lambda h.\lambda t.\ t]\!]senv = \langle ABS, \lambda\hat{h}.\langle ABS, \lambda\hat{t}.\langle T \sqcap \hat{t}_v, serr\rangle\rangle\rangle$$

We can formalize all this above.

Non-Standard Semantic Domains

V,	variables of interest
V_C,	finite domain of projections over V
	for example if $V = INT$, V_C might be the two – point domain {STR, ID}
$LIST\ V$,	non – flat domain of lists whose elements are in domain V
$LIST_C\ V_C$,	finite domain of projections over LIST V
$SP = LIST_C\ V_C \times (SP \rightarrow SP)$,	domain of strictness pairs
$Senv = V \rightarrow SP$,	the strictness environments

We also define a special error element $serr = \lambda\hat{x}.\ \langle FAIL, serr\rangle$ to be used when expression has been applied "too many times". We write $e_1 \& e_2$ to mean $\lambda\hat{x}.\ \langle (e_1\ \hat{x})_v\ \&\ (e_2\ \hat{x})_v, (e_1\ \hat{x})_f\ \&\ (e_2\ \hat{x})_f\rangle$, where & between two projections has been defined in the previous section.

Non-Standard Semantic Functions

$\mathcal{K}_s$:	$Con \rightarrow SP$,	maps constants to SP
$\mathcal{S}$:	$Exp \rightarrow Senv \rightarrow SP$,	maps expressions to strictness pairs
$\mathcal{P}_s$:	$Prog \rightarrow Senv$,	gives meaning to programs

$$
\begin{array}{rcl}
\mathcal{K}_s[\![c]\!] & = & \langle ABS, serr\rangle \\
\mathcal{K}_s[\![+]\!] & = & \langle ABS, \lambda\hat{x}.\langle ABS, \lambda\hat{y}.\langle \hat{x}_v \& \hat{y}_v, serr\rangle\rangle\rangle \\
\mathcal{K}_s[\![=]\!] & = & \mathcal{K}_s[\![+]\!] \\
\mathcal{K}_s[\![null?]\!] & = & \langle ABS, \lambda\hat{xs}.\langle \hat{xs}_v, serr\rangle\rangle \\
\mathcal{K}_s[\![Nil]\!] & = & \langle ABS, serr\rangle \\
 & & \\
\mathcal{S}[\![k]\!]senv & = & \mathcal{K}_s[\![k]\!] \\
\mathcal{S}[\![x]\!]senv & = & senv[\![x]\!] \\
\mathcal{S}[\![e_1 \rightarrow e_2, e_3]\!]senv & = & \langle (\mathcal{S}[\![e_1]\!]senv)_v \& ((\mathcal{S}[\![e_2]\!]senv)_v \sqcup (\mathcal{S}[\![e_3]\!]senv)_v), (\mathcal{S}[\![e_2]\!]senv)_f \& (\mathcal{S}[\![e_3]\!]senv)_f\rangle \\
\mathcal{S}[\![\lambda x.e]\!]senv & = & \langle ABS, \lambda\hat{x}.\ \mathcal{S}[\![e]\!]senv[\hat{x}/x]\rangle \\
\mathcal{S}[\![e_1\ e_2]\!]senv & = & \langle (\mathcal{S}[\![e_1]\!]senv)_v \& sv, sf\rangle \\
 & & \text{where } \langle sv, sf\rangle = (\mathcal{S}[\![e_1]\!]senv)_f\ (\mathcal{S}[\![e_2]\!]senv) \\
\mathcal{S}[\![\lambda h.\lambda t.\ h]\!]senv & = & \langle ABS, \lambda\hat{h}.\langle H \sqcap \hat{h}_v, \lambda\hat{t}.\langle ABS, serr\rangle\rangle\rangle \\
\mathcal{S}[\![\lambda h.\lambda t.\ t]\!]senv & = & \langle ABS, \lambda\hat{h}.\langle ABS, \lambda\hat{t}.\langle T \sqcap \hat{t}_v, serr\rangle\rangle\rangle \\
\mathcal{S}[\![\{f_1 = e_1, ..., f_n = e_n\}]\!]senv & = & \mathcal{S}[\![e_1]\!]senv' \\
 & & \text{whererec } senv' = senv[\mathcal{S}[\![e_1]\!]senv'/f_1, ..., \mathcal{S}[\![e_n]\!]senv'/f_n] \\
\mathcal{P}_s[\![\{f_1 = e_1, ..., f_n = e_n\}]\!]senv & = & senv' \\
 & & \text{whererec } senv' = senv[\mathcal{S}[\![e_1]\!]senv'/f_1, ..., \mathcal{S}[\![e_n]\!]senv'/f_n]
\end{array}
$$

Examples

To see how this analysis works, let us consider two examples, *length* and *before*.

Let us define $f = length$ and $g_1 = g\ tt$, where $tt = \lambda h.\lambda t.t$. So *length* can be rewritten as

$$f = \lambda g.null?\ g \rightarrow 0, +1\ (f\ g_1)$$

For clarity, we will write $\hat{f} = \mathcal{S}[\![f]\!]senv$ and we omit the environment argument to $\mathcal{S}$ in all cases.

$$
\begin{array}{rcl}
\hat{f} & = & \langle ABS, \lambda\hat{g}.\mathcal{S}[\![null?g \rightarrow 0,\ +1\ (f\ g_1)]\!]\rangle \\
\hat{f} & = & \langle ABS, \lambda\hat{g}.\langle \mathcal{S}_v[\![null?g]\!]\ \&\ (\mathcal{S}_v[\![0]\!] \sqcup \mathcal{S}_v[\![+1\ (f\ g_1)]\!]), \mathcal{S}_f[\![0]\!]\ \&\ \mathcal{S}_f[\![+1\ (fg_1)]\!]\rangle\rangle \\
\mathcal{S}[\![0]\!] & = & \mathcal{K}_s[\![0]\!] = \langle ABS, serr\rangle\ (recall\ FAIL\&\alpha = FAIL) \\
\hat{f} & = & \langle ABS, \lambda\hat{g}.\langle \mathcal{S}_v[\![null?g]\!]\ \&\ (ABS \sqcup \mathcal{S}_v[\![+1\ (f\ g_1)]\!]), serr\rangle\rangle \\
\mathcal{S}[\![null?g]\!] & = & \langle \mathcal{S}[\![null?]\!]_v \& sv_1, sf_1\rangle \\
 & & where\ \langle sv_1, sf_1\rangle = (\mathcal{S}_f[\![null?]\!])(\hat{g}) \\
\mathcal{S}[\![null?]\!] & = & \mathcal{K}_s[\![null?]\!] = \langle ABS, \lambda\hat{x}\hat{s}.\langle \hat{x}\hat{s}_v, serr\rangle\rangle \\
 & & \langle sv_1, sf_1\rangle = (\lambda\hat{x}\hat{s}.\langle \hat{x}\hat{s}_v, serr\rangle)(\hat{g}) = \langle \hat{g}_v, serr\rangle \\
\mathcal{S}[\![null?g]\!] & = & \langle ABS \& \hat{g}_v, serr\rangle = \langle \hat{g}_v, serr\rangle \\
\hat{f} & = & \langle ABS, \lambda\hat{g}.\langle \hat{g}_v\ \&\ (ABS \sqcup \mathcal{S}_v[\![+1\ (f\ g_1)]\!]), serr\rangle\rangle \\
\mathcal{S}[\![+1\ (fg_1)]\!] & = & \langle \mathcal{S}_v[\![+1]\!] \& sv_1, sf_1\rangle \\
 & & where\ \langle sv_1, sf_1\rangle = (\mathcal{S}_f[\![+1]\!])(\mathcal{S}[\![fg_1]\!]) \\
\mathcal{S}[\![+\ 1]\!] & = & \langle \mathcal{S}_v[\![+\ 1]\!] \& sv_2, sf_2\rangle \\
 & & where\ \langle sv_2, sf_2\rangle = (\mathcal{S}_f[\![+]\!])(\mathcal{S}[\![1]\!]) \\
\mathcal{S}[\![+]\!] & = & \mathcal{K}_s[\![+]\!] = \langle ABS, \lambda\hat{x}.\langle ABS, \lambda\hat{y}.\langle \hat{x}_v \& \hat{y}_v, serr\rangle\rangle\rangle \\
\mathcal{S}[\![1]\!] & = & \mathcal{K}_s[\![1]\!] = \langle ABS, serr\rangle \\
 & & \langle sv_2, sf_2\rangle = (\lambda\hat{x}.\langle ABS, \lambda\hat{y}.\langle \hat{x}_v \& \hat{y}_v, serr\rangle\rangle)(\mathcal{K}_s[\![1]\!]) \\
 & & \langle sv_2, sf_2\rangle = \langle ABS, \lambda\hat{y}.\langle \mathcal{K}_{sv}[\![1]\!] \& \hat{y}_v, serr\rangle\rangle \\
 & & \langle sv_2, sf_2\rangle = \langle ABS, \lambda\hat{y}.\langle ABS \& \hat{y}_v, serr\rangle\rangle \\
 & & \langle sv_2, sf_2\rangle = \langle ABS, \lambda\hat{y}.\langle \hat{y}_v, serr\rangle\rangle \\
\mathcal{S}[\![+1]\!] & = & \langle ABS \& ABS, \lambda\hat{y}.\langle \hat{y}_v, serr\rangle\rangle \\
\mathcal{S}[\![+1]\!] & = & \langle ABS, \lambda\hat{y}.\langle \hat{y}_v, serr\rangle\rangle \\
 & & \langle sv_1, sf_1\rangle = (\lambda\hat{y}.\langle \hat{y}_v, serr\rangle)(\mathcal{S}[\![fg_1]\!]) \\
 & & \langle sv_1, sf_1\rangle = (\langle \mathcal{S}_v[\![fg_1]\!], serr\rangle \\
\mathcal{S}[\![+1\ (fg_1)]\!] & = & \langle ABS \& \mathcal{S}_v[\![fg_1]\!], serr\rangle \\
\mathcal{S}[\![+1\ (fg_1)]\!] & = & \langle \mathcal{S}_v[\![fg_1]\!], serr\rangle \\
\hat{f} & = & \langle ABS, \lambda\hat{g}.\langle \hat{g}_v\ \&\ (ABS \sqcup \mathcal{S}_v[\![f\ g_1]\!]), serr\rangle\rangle \\
\mathcal{S}[\![f\ g_1]\!] & = & \langle \mathcal{S}_v[\![f]\!] \& sv, sf\rangle \\
 & & where\ \langle sv, sf\rangle = (\mathcal{S}_f[\![f]\!])(\hat{g}_1) \\
\mathcal{S}[\![f\ g_1]\!] & = & \langle \hat{f}_v \& (\hat{f}_f\ \hat{g}_1)_v\ ,\ (\hat{f}_f\ \hat{g}_1)_f\rangle \\
\hat{f} & = & \langle ABS, \lambda\hat{g}.\langle \hat{g}_v \& (ABS \sqcup (\hat{f}_v \& (\hat{f}_f g_1)_v)), serr\rangle\rangle \\
 & & \\
\hat{g}_1 & = & \mathcal{S}[\![g_1]\!] = \mathcal{S}[\![g\ tt]\!] = \langle \mathcal{S}_v[\![g]\!] \& sv, sf\rangle \\
 & & where\ \langle sv, sf\rangle = (\mathcal{S}_f[\![g]\!])(\mathcal{S}[\![tt]\!]) \\
\hat{g}_1 & = & \langle \hat{g}_v \& (\hat{g}_f\ \hat{tt})_v, (\hat{g}_f\ \hat{tt})_f\rangle \quad where\ tt = \lambda h.\lambda t.t \\
\hat{tt} & = & \mathcal{S}[\![\lambda h.\lambda t.\ t]\!] = \langle ABS, \lambda\hat{h}.\langle ABS, \lambda\hat{t}.\langle T \sqcap \hat{t}_v, serr\rangle\rangle\rangle \\
g & = & \lambda c.c\ x_1\ g_1 \quad where\ x_1 = g\ hh\ and\ g_1 = g\ tt \\
 & & \\
\hat{g} & = & \langle ABS, \lambda\hat{c}.\mathcal{S}[\![c\ x_1\ g_1]\!]\rangle \\
 & & \\
\mathcal{S}[\![c\ x_1\ g_1]\!] & = & \langle \mathcal{S}_v[\![c\ x_1]\!] \& sv1, sf1\rangle \\
 & & where\ \langle sv_1, sf_1\rangle = (\mathcal{S}_f[\![c\ x_1]\!])(\hat{g}_1) \\
\mathcal{S}[\![c\ x_1]\!] & = & \langle \mathcal{S}[\![c]\!] \& sv_2\rangle \\
 & & where\ \langle sv_2, sf_2\rangle = (\mathcal{S}_f[\![c]\!])(\hat{x}_1) \\
\hat{g} & = & \langle ABS, \lambda\hat{c}.\langle \hat{c}_v \& (\hat{c}_f\hat{x}_1)_v \& ((\hat{c}_f\hat{x}_1)_f\hat{g}_1)_v, ((\hat{c}_f\hat{x}_1)_f\hat{g}_1)_f\rangle\rangle \\
\hat{g}_f\hat{tt} & = & (\lambda\hat{c}.\langle \hat{c}_v \& (\hat{c}_f\hat{x}_1)_v \& ((\hat{c}_f\hat{x}_1)_f\hat{g}_1)_v, ((\hat{c}_f\hat{x}_1)_f\hat{g}_1)_f\rangle)(\hat{tt}) \\
\hat{g}_f\hat{tt} & = & \langle \hat{tt}_v \& (\hat{tt}_f\hat{x}_1)_v \& ((\hat{tt}_f\hat{x}_1)_f\hat{g}_1)_v, ((\hat{tt}_f\hat{x}_1)_f\hat{g}_1)_f\rangle \\
\hat{g}_f\hat{tt} & = & \langle ABS \& (ABS) \& ((\lambda\hat{t}.\langle \hat{t}_v \sqcap T, serr\rangle)\hat{g}_1)_v, ((\lambda\hat{t}.\langle \hat{t}_v \sqcap T, serr\rangle)\hat{g}_1)_f\rangle \\
\hat{g}_f\ \hat{tt} & = & \langle T \sqcap (\hat{g}_1)_v, serr\rangle \\
\hat{g}_1 & = & \langle \hat{g}_v \& (T \sqcap (\hat{g}_1)_v), serr\rangle \\
 & & \\
\hat{f} & = & \langle ABS, \lambda\hat{g}.\langle \hat{g}_v \& (ABS \sqcup (\hat{f}_v \& (\hat{f}_f\hat{g}_1)_v)), serr\rangle\rangle \\
where\ \hat{g}_1 & = & \langle \hat{g}_v \& (T \sqcap (\hat{g}_1)_v), serr\rangle
\end{array}
$$

We get a recursive function for $\hat{f}$. So we have to compute the least fixpoint by constructing Kleene's ascending chain of approximations. What we desire to get as a solution is that f is tail-strict. AKC construction of approximations starts with the bottom element in the lattice of

$$SP = LIST_C\ D_C \times (SP \rightarrow SP)$$

i.e. $\perp_{SP} = \langle FAIL, serr \rangle$. While analysing the strictness property of f, we have to take into account the strictness properties of g and $g_1 = g\ tt$. As intuitively inferred, we see that $\hat{f}$ depends on $(\hat{f}_f\ \hat{g}_1)_v$. It means that f needs to evaluate the entire spine of its arguments, i.e. all the tails, so f is tail-strict. We have to compute also the least fixpoint of $\hat{g}_1$ and the AKC construction of approximations starts with $\langle STR, serr \rangle$.

$$\begin{aligned}
\hat{g}_1^{(0)} &= \langle STR, serr \rangle \\
\hat{g}_1^{(1)} &= \langle \hat{g}_v \& (T \sqcap STR), serr \rangle \\
\hat{g}_1^{(2)} &= \langle \hat{g}_v \& (T \sqcap (\hat{g}_v \& (T \sqcap STR))), serr \rangle \\
\hat{g}_1^{(2)} &= \langle \hat{g}_v \& (T \sqcap STR), serr \rangle \\
&\cdots \\
\hat{g}_1 &= \langle \hat{g}_v \& (T \sqcap STR), serr \rangle
\end{aligned}$$

$$\begin{aligned}
\hat{f}^{(0)} &= \langle FAIL, serr \rangle \\
\hat{f}^{(1)} &= \langle ABS, \lambda \hat{g}. \langle \hat{g}_v \& (ABS \sqcup (FAIL \& (\hat{f}_f \hat{g}_1)_v)), serr \rangle \rangle \\
\hat{f}^{(1)} &= \langle ABS, \lambda \hat{g}. \langle \hat{g}_v \& (ABS \sqcup FAIL), serr \rangle \rangle \\
\hat{f}^{(1)} &= \langle ABS, \lambda \hat{g}. \langle \hat{g}_v \& ABS, serr \rangle \rangle \\
\hat{f}^{(1)} &= \langle ABS, \lambda \hat{g}. \langle \hat{g}_v, serr \rangle \rangle
\end{aligned}$$

$$\begin{aligned}
\hat{f}^{(2)} &= \langle ABS, \lambda \hat{g}. \langle \hat{g}_v \& (ABS \sqcup (ABS \& ((\lambda \hat{g}. \langle \hat{g}_v, serr \rangle) \hat{g}_1)_v)), serr \rangle \rangle \\
\hat{f}^{(2)} &= \langle ABS, \lambda \hat{g}. \langle \hat{g}_v \& (ABS \sqcup (\hat{g}_1)_v), serr \rangle \rangle \\
\hat{f}^{(2)} &= \langle ABS, \lambda \hat{g}. \langle \hat{g}_v \& (\hat{g}_1)_v, serr \rangle \rangle \\
\hat{f}^{(2)} &= \langle ABS, \lambda \hat{g}. \langle \hat{g}_v \& (\hat{g}_v \& (T \sqcap STR)), serr \rangle \rangle \\
\hat{f}^{(2)} &= \langle ABS, \lambda \hat{g}. \langle \hat{g}_v \& (T \sqcap STR), serr \rangle \rangle
\end{aligned}$$

$$\begin{aligned}
\hat{f}^{(3)} &= \langle ABS, \lambda \hat{g}. \langle \hat{g}_v \& (ABS \sqcup (ABS \& ((\lambda \hat{g}. \langle \hat{g}_v \& (T \sqcap STR), serr \rangle) \hat{g}_1)_v)), serr \rangle \rangle \\
\hat{f}^{(3)} &= \langle ABS, \lambda \hat{g}. \langle \hat{g}_v \& (ABS \sqcup ((\hat{g}_1)_v \& (T \sqcap STR))), serr \rangle \rangle \\
\hat{f}^{(3)} &= \langle ABS, \lambda \hat{g}. \langle \hat{g}_v \& (\hat{g}_1)_v \& (T \sqcap STR), serr \rangle \rangle \\
\hat{f}^{(3)} &= \langle ABS, \lambda \hat{g}. \langle \hat{g}_v \& (\hat{g}_v \& (T \sqcap STR)) \& (T \sqcap STR), serr \rangle \rangle \\
\hat{f}^{(3)} &= \langle ABS, \lambda \hat{g}. \langle \hat{g}_v \& (T \sqcap STR), serr \rangle \rangle \\
&\cdots \\
\hat{f} &= \langle ABS, \lambda \hat{g}. \langle \hat{g}_v \& (T \sqcap STR), serr \rangle \rangle
\end{aligned}$$

So f is tail-strict.

Now let us define $f = before$, $x_1 = g\ hh$ and $g_1 = g\ tt$, where $hh = \lambda h. \lambda t. h$ and $tt = \lambda h. \lambda t. t$. So $before$ can be rewritten as

$$\begin{aligned}
f &= \lambda g. null?\ g \rightarrow Nil,\ ((=\ (g\ hh)\ 0) \rightarrow Nil, \lambda c.\ c (g\ hh) (f\ (g\ tt))) \\
f &= \lambda g. null?\ g \rightarrow Nil,\ ((=\ x_1\ 0) \rightarrow Nil, \lambda c.\ c\ x_1\ (f\ g_1))
\end{aligned}$$

$$
\begin{aligned}
\hat{f} &= \langle ABS, \lambda\hat{g}.\mathcal{S}[\![null?g \to Nil,\ ((=\ x_1\ 0) \to Nil, \lambda c.\ c\ x_1\ (f\ g_1))]\!]\rangle \\
\hat{f} &= \langle ABS, \lambda\hat{g}.\langle \mathcal{S}_v[\![null?g]\!]\ \&\ (\mathcal{S}_v[\![Nil]\!] \sqcup \mathcal{S}_v[\![(=\ x_1\ 0) \to Nil, \lambda c.\ c\ x_1\ (f\ g_1)]\!]), \\
&\quad \mathcal{S}_f[\![Nil]\!]\ \&\ \mathcal{S}_f[\![(=\ x_1\ 0) \to Nil, \lambda c.\ c\ x_1\ (f\ g_1)]\!]\rangle\rangle \\
\mathcal{S}[\![null?g]\!] &= \langle \hat{g}_v, serr\rangle \\
\mathcal{S}[\![Nil]\!] &= \mathcal{K}_s[\![Nil]\!] = \langle ABS, serr\rangle \\
\hat{f} &= \langle ABS, \lambda\hat{g}.\langle \hat{g}_v\ \&\ (ABS \sqcup \mathcal{S}_v[\![(=\ x_1\ 0) \to Nil, \lambda c.\ c.x_1\ (f\ g_1)]\!]), serr\rangle\rangle \\
\mathcal{S}[\![(=\ x_1\ 0) \to Nil, \lambda c.c\ x_1\ (f\ g_1)]\!] &= \langle \mathcal{S}_v[\![=\ x_1\ 0]\!]\ \&\ (\mathcal{S}_v[\![Nil]\!] \sqcup \mathcal{S}_v[\![\lambda c.c\ x_1\ (f\ g_1)]\!]), \mathcal{S}_f[\![Nil]\!]\ \&\ \mathcal{S}_f[\![\lambda c.c\ x_1\ (fg_1)]\!]\rangle \\
\mathcal{S}[\![(=\ x_1\ 0) \to Nil, \lambda c.c\ x_1\ (f\ g_1)]\!] &= \langle \mathcal{S}_v[\![=\ x_1\ 0]\!]\ \&\ (ABS \sqcup \mathcal{S}_v[\![\lambda c.c\ x_1\ (f\ g_1)]\!]), serr\rangle \\
\\
\mathcal{S}[\![=\ x_1\ 0]\!] &= \langle \mathcal{S}_v[\![=\ x_1]\!] \& sv_1, sf_1\rangle \\
&\quad where\ \langle sv_1, sf_1\rangle = (\mathcal{S}_f[\![=\ x_1]\!])(\mathcal{S}[\![0]\!]) \\
\mathcal{S}[\![=\ x_1]\!] &= \langle \mathcal{S}[\![=]\!]_v \& sv_2, sf_2\rangle \\
&\quad where\ \langle sv_2, sf_2\rangle = (\mathcal{S}_f[\![=]\!])(\mathcal{S}[\![x_1]\!]) \\
\mathcal{S}[\![=]\!] &= \mathcal{K}_s[\![=]\!] = \langle ABS, \lambda\hat{x}.\langle ABS, \lambda\hat{y}.\langle \hat{x}_v \& \hat{y}_v, serr\rangle\rangle\rangle \\
&\quad \langle sv_2, sf_2\rangle = (\lambda\hat{x}.\langle ABS, \lambda\hat{y}.\langle \hat{x}_v \& \hat{y}_v, serr\rangle\rangle)(\mathcal{S}[\![x_1]\!]) \\
&\quad \langle sv_2, sf_2\rangle = \langle ABS, \lambda\hat{y}.\langle (\hat{x}_1)_v \& \hat{y}_v, serr\rangle\rangle \\
\mathcal{S}[\![=\ x_1]\!] &= \langle ABS \& ABS, \lambda\hat{y}.\langle (\hat{x}_1)_v \& \hat{y}_v, serr\rangle\rangle \\
&\quad \langle sv_1, sf_1\rangle = (\lambda\hat{y}.\langle (\hat{x}_1)_v \& \hat{y}_v, serr\rangle)(\mathcal{S}[\![0]\!]) \\
&\quad \langle sv_1, sf_1\rangle = \langle (\hat{x}_1)_v \& \mathcal{S}[\![0]\!], serr\rangle \\
\mathcal{S}[\![0]\!] &= \mathcal{K}_s[\![0]\!] = \langle ABS, serr\rangle \\
&\quad \langle sv_1, sf_1\rangle = \langle (\hat{x}_1)_v \& ABS, serr\rangle = \langle (\hat{x}_1)_v\ serr\rangle \\
\mathcal{S}[\![=\ x_1\ 0]\!] &= \langle ABS \& (\hat{x}_1)_v, serr\rangle \\
\mathcal{S}[\![\lambda c.c\ x_1\ (f\ g_1)]\!] &= \langle ABS, \lambda\hat{c}.\mathcal{S}[\![c\ x_1\ g_1]\!]\rangle \\
\mathcal{S}[\![(=\ x_1\ 0) \to Nil, \lambda c.c\ x_1\ (f\ g_1)]\!] &= \langle (\hat{x}_1)_v \& (ABS \sqcup ABS), serr\rangle \\
\hat{f} &= \langle ABS, \lambda\hat{g}.\langle \hat{g}_v \& (ABS \sqcup (\hat{x}_1)_v), serr\rangle\rangle \\
\\
\hat{x}_1 &= \mathcal{S}[\![x_1]\!] = \mathcal{S}[\![g\ hh]\!] = \langle \mathcal{S}_v[\![g]\!] \& sv, sf\rangle \\
&\quad where\ \langle sv, sf\rangle = (\mathcal{S}_f[\![g]\!])(\mathcal{S}[\![hh]\!]) \\
\hat{x}_1 &= \langle \hat{g}_v \& (\hat{g}_f\ \hat{hh})_v, (\hat{g}_f\ \hat{hh})_f\rangle \quad where\ hh = \lambda h.\lambda t.h \\
\\
\hat{f} &= \langle ABS, \lambda\hat{g}.\langle \hat{g}_v \& (ABS \sqcup (\hat{g}_v \& (\hat{g}_f\ \hat{hh})_v), serr\rangle\rangle \\
\\
\hat{hh} &= \mathcal{S}[\![\lambda h.\lambda t.\ h]\!] = \langle ABS, \lambda\hat{h}.\langle H \sqcap \hat{h}, \lambda\hat{t}.\langle ABS, serr\rangle\rangle\rangle \\
g &= \lambda c.c\ x_1\ g_1 \quad where\ x_1 = g\ hh\ and\ g_1 = g\ tt \\
\hat{g} &= \langle ABS, \lambda\hat{c}.\langle \hat{c}_v \& (\hat{c}_f\hat{x}_1)_v \& ((\hat{c}_f\hat{x}_1)_f\hat{g}_1)_v, ((\hat{c}_f\hat{x}_1)_f\hat{g}_1)_f\rangle\rangle \\
\hat{g}_f\hat{hh} &= (\lambda\hat{c}.\langle \hat{c}_v \& (\hat{c}_f\hat{x}_1)_v \& ((\hat{c}_f\hat{x}_1)_f\hat{g}_1)_v, ((\hat{c}_f\hat{x}_1)_f\hat{g}_1)_f\rangle)(\hat{hh}) \\
\hat{g}_f\hat{hh} &= \langle \hat{hh}_v \& (\hat{hh}_f\hat{x}_1)_v \& ((\hat{hh}_f\hat{x}_1)_f\hat{g}_1)_v, ((\hat{hh}_f\hat{x}_1)_f\hat{g}_1)_f\rangle \\
\hat{g}_f\hat{hh} &= \langle ABS \& (H \sqcap (\hat{x}_1)_v) \& ((\lambda\hat{t}.\langle ABS, serr\rangle)\hat{g}_1)_v, ((\lambda\hat{t}.\langle ABS, serr\rangle)\hat{g}_1)_f\rangle \\
\hat{g}_f\ \hat{hh} &= \langle H \sqcap (\hat{x}_1)_v \& ABS, serr\rangle = \langle H \sqcap (\hat{x}_1)_v, serr\rangle \\
\\
\hat{x}_1 &= \langle \hat{g}_v \& (H \sqcap (\hat{x}_1)_v), serr\rangle \\
\\
\hat{f} &= \langle ABS, \lambda\hat{g}.\langle \hat{g}_v \& (ABS \sqcup (\hat{g}_v \& (H \sqcap (\hat{x}_1)_v))), serr\rangle\rangle \\
&\quad where\ \hat{x}_1 = \langle \hat{g}_v \& (H \sqcap (\hat{x}_1)_v), serr\rangle
\end{aligned}
$$

What we desire to get as a solution is that f is head-strict. We get an equation saying that the strictness property of f depends on the stricness properties of g and $x_1 = g\ hh$. It means that the first thing that f does is to check its first element, so f is head-strict. We get a recursive equation of x_1. So we have to compute the least fixpoint by

constructing AKC of approximations starting with $\langle STR, serr\rangle$.

$$\begin{aligned}
\hat{x}_1^{(0)} &= \langle STR, serr\rangle \\
\hat{x}_1^{(1)} &= \langle \hat{g}_v\&(H \sqcap STR), serr\rangle \\
\hat{x}_1^{(2)} &= \langle \hat{g}_v\&(H \sqcap (\hat{g}_v\&(H \sqcap STR))), serr\rangle \\
\hat{x}_1^{(2)} &= \langle \hat{g}_v\&(H \sqcap STR), serr\rangle \\
&\cdots \\
\hat{x}_1 &= \langle \hat{g}_v\&(H \sqcap STR), serr\rangle
\end{aligned}$$

$$\begin{aligned}
\hat{f} &= \langle ABS, \lambda\hat{g}.\langle \hat{g}_v\&(ABS \sqcup (\hat{g}_v\&(H \sqcap STR))), serr\rangle\rangle \\
\hat{f} &= \langle ABS, \lambda\hat{g}.\langle \hat{g}_v\&(\hat{g}_v\&(H \sqcap STR)), serr\rangle\rangle \\
\hat{f} &= \langle ABS, \lambda\hat{g}.\langle \hat{g}_v\&(H \sqcap STR), serr\rangle\rangle
\end{aligned}$$

So f is head-strict.

This simple approach allows us to get the strictness properties that we desire. But, in order to do that we have introduced some syntactic restrictions in our analysis by favouring two specific functions, hh and tt. In order to get a generalized method we will try first to lift these syntactic restrictions and secondly to deal with a generalized notion of strictness analysis for other kind of data structures.

4 Generalization of the method

In the previous section we have considered lists which can be decomposed recursively in two components: a head and a tail. We were interested in the head and tail strictness properties of such a structure , which were formalized by the projections H and T combined with the strictness properties of the arguments.

Let us now consider the following function $\lambda x.\lambda y.\lambda z.y$, the curried form of a data structure. What we desire to say about this function is that it is strict in the second component, i.e.

$$\begin{aligned}
\mathcal{S}[\![\lambda x.\lambda y.\lambda z.\ y]\!]senv &= \langle ABS, \lambda\hat{x}.\langle ABS, \lambda\hat{y}.\langle ABS, \lambda\hat{z}.\langle\overline{2}, serr\rangle\rangle\rangle\rangle \\
&\quad where\ \overline{2} = (ABS, \hat{y}, ABS).
\end{aligned}$$

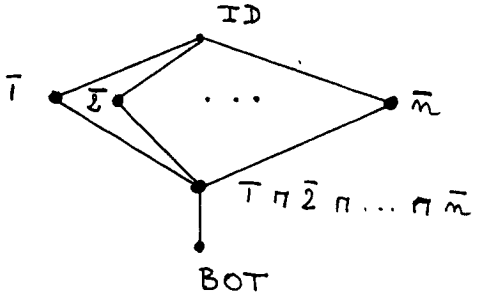
This notion of strictness of the n-th component of a data structure should be represented by such a formalism.

$$\begin{aligned}
\overline{1} &= (\hat{x_1}, ABS, ..., ABS) \\
\overline{2} &= (ABS, \hat{x_2}, ..., ABS) \\
&\cdots \\
\overline{n} &= (ABS, ..., ABS, \hat{x_n})
\end{aligned}$$

For example, H and T can be represented by

$$\begin{aligned}
H = \overline{1} &= (\hat{h}, ABS) \\
T = \overline{2} &= (ABS, \hat{t})
\end{aligned}$$

In order to capture the behaviour of other kind of data structures, we have to extend the finite domain of projections, that we used in the construction of the domain of strictness pairs. If we restrict ourselves to a the Hindley-Milner type system, which does not allow infinite recursive type, the domain of projections is still finite. As in section 2 for the projections H and T, we can represent $\overline{1}, \overline{2}, ...\overline{n}$ as projection of the following lattice.



In the above example, we get $\theta = \{3 : x \mapsto \hat{x}, 2 : y \mapsto \hat{y}, 1 : z \mapsto \hat{z}\}$.

With this labelling, we can define

$$\begin{array}{rcl} \overline{y} = \theta(\mathcal{S}[\![y]\!]senv) & = & (\theta_3(\mathcal{S}[\![y]\!]senv), \theta_2(\mathcal{S}[\![y]\!]senv), \theta_1(\mathcal{S}[\![y]\!]senv)) \\ with\ \theta_i(\mathcal{S}[\![y]\!]senv) & = & senv[\![y]\!]\ if\ \exists i\ /\ i : y \mapsto \hat{y} \\ & = & ABS,\ otherwise \\ \\ so\ \overline{y} & = & (ABS, \hat{y}, ABS) \end{array}$$

In order to capture this behaviour, we need to add to the strictness environment, Senv, some new objects

$$\begin{array}{rcl} Numb & = & Label \times V \to V_C, \\ Count & = & Numb^n \to DS_C\ V_C, \\ Senv & = & V_C \to Numb^n \to SP, \end{array}$$

where *Count* collects all the numbered bindings of variables updated in the strictness environments of a data structure. In the previous example, $\theta \in Count = \{3 : x \mapsto \hat{x}, 2 : y \mapsto \hat{y}, 1 : z \mapsto \hat{z}\}$. The formal definition of this new object is as follows

$$\begin{array}{rcl} \theta : Count = Numb^n & \to & DS_C\ V_C \\ \mathcal{S}[\![x]\!]senv & \mapsto & \theta(\mathcal{S}[\![x]\!]senv) \\ \forall\ \theta_i : Numb \in \theta,\ i = 1,n & & \\ where\ \theta(\mathcal{S}[\![x]\!]senv) & = & (\theta_n(\mathcal{S}[\![x]\!]senv), ..., \theta_1(\mathcal{S}[\![x]\!]senv)) \\ with\ \theta_i(\mathcal{S}[\![x]\!]senv) & = & senv[\![x]\!]\ if\ \exists i\ /\ i : x \mapsto \hat{x} \\ & = & ABS,\ otherwise \end{array}$$

We can formalize all above.

Non-Standard Semantic Domains

V,	variables of interest
V_C,	finite domain of projections over V
$DS\ V$,	non – flat domain of data structures whose elements are in domain V
$DS_C\ V_C$,	finite domain of projections over DS V
$SP = DS_C\ V_C \times (SP \to SP)$,	domain of strictness pairs
$Numb = Label \times V \to V_C$	
$Count = Numb^n \to DS_C\ V_C$	
$Senv = V_C \to Numb^n \to SP$,	the strictness environments

Non-Standard Semantic Functions

$\mathcal{K}_s$:	$Con \to SP$,	maps constants to SP assumed given
$\mathcal{S}$:	$Exp \to Senv \to SP$,	maps expressions to strictness pairs
$\mathcal{P}_s$:	$Prog \to SP$,	gives meaning to programs

$$\begin{array}{rcl} \theta_{init} & = & \emptyset \\ \mathcal{S}[\![k]\!]senv\theta & = & \mathcal{K}_s[\![k]\!] \\ \mathcal{S}[\![x]\!]senv\theta & = & \langle \theta(senv[\![x]\!]), serr \rangle \\ \mathcal{S}[\![e_1 \to e_2, e_3]\!]senv\theta & = & \langle (\mathcal{S}[\![e_1]\!]senv\theta)_v \& ((\mathcal{S}[\![e_2]\!]senv\theta)_v \sqcup (\mathcal{S}[\![e_3]\!]senv\theta)_v), \\ & & (\mathcal{S}[\![e_2]\!]senv\theta)_f \& (\mathcal{S}[\![e_3]\!]senv\theta)_f \rangle \\ \mathcal{S}[\![\lambda x_{lab}.e]\!]senv\theta & = & \langle ABS, \lambda\hat{x}.\ \mathcal{S}[\![e]\!]senv[\hat{x}/x]\theta' \rangle \\ & & where\ \theta' = \theta \cup \{lab : x \mapsto \hat{x}\} \\ \mathcal{S}[\![e_1\ e_2]\!]senv\theta & = & \langle (\mathcal{S}[\![e_1]\!]senv\emptyset)_v \& sv, sf \rangle \\ & & \text{where } \langle sv, sf \rangle = (\mathcal{S}[\![e_1]\!]senv\emptyset)_f\ (\mathcal{S}[\![e_2]\!]senv\emptyset) \\ \mathcal{S}[\![\{f_1 = e_1, ..., f_n = e_n\}]\!]senv\theta & = & \mathcal{S}[\![e_1]\!]senv'\theta \\ & & \text{whererec } senv' = senv[\mathcal{S}[\![e_1]\!]senv'/f_1, ..., \mathcal{S}[\![e_n]\!]senv'/f_n] \\ \mathcal{P}_s[\![\{f_1 = e_1, ..., f_n = e_n\}]\!]senv\theta & = & senv'\theta \\ & & \text{whererec } senv' = senv[\mathcal{S}[\![e_1]\!]senv'/f_1, ..., \mathcal{S}[\![e_n]\!]senv'/f_n] \end{array}$$

Example

To see how this analysis works, let us consider the example of $f = length$. So *length* can be rewritten as

$$f = \lambda g_3.null?\ g \rightarrow 0,\ +1\ (f\ (g\ (\lambda h_2.\lambda t_1.t)))$$

For clarity, we will write $\hat{f} = \mathcal{S}[\![f]\!]senv$ and we omit the environment argument to $\mathcal{S}$ in all cases. We omit the θ argument when it is $\emptyset$.

$$\begin{array}{rcl}
\theta & = & \emptyset \\
\hat{f}\theta & = & \langle ABS, \lambda\hat{g}.\mathcal{S}[\![null?g \rightarrow 0,\ +1\ (f\ g\ (\lambda h_2.\lambda t_1.t)))]\!]\theta'\rangle \\
 & & where\ \theta' = \theta \sqcup \{3 : g \mapsto \hat{g}\} = \{3 : g \mapsto \hat{g}\} \\
\hat{f}\theta & = & \langle ABS, \lambda\hat{g}.\langle \mathcal{S}_v[\![null?g]\!]\theta' \ \&\ (\mathcal{S}_v[\![0]\!]\theta' \sqcup \mathcal{S}_v[\![+1\ (f\ g\ (\lambda h_2.\lambda t_1.t)))]\!]\theta'), \\
 & & \mathcal{S}_f[\![0]\!]\theta' \ \&\ \mathcal{S}_f[\![+1\ (fg\ (\lambda h_2.\lambda t_1.t)))]\!]\theta'\rangle\rangle \\
 & & where\ \theta' = \{3 : g \mapsto \hat{g}\} \\
\mathcal{S}[\![0]\!]\theta' & = & \mathcal{K}_s[\![0]\!] = \langle ABS, serr\rangle\ (recall\ FAIL\&\alpha = FAIL) \\
\hat{f}\theta & = & \langle ABS, \lambda\hat{g}.\langle \mathcal{S}_v[\![null?g]\!]\theta' \ \&\ (ABS \sqcup \mathcal{S}_v[\![+1\ (f\ g\ (\lambda h_2.\lambda t_1.t)))]\!]\theta'), serr\rangle\rangle \\
 & & where\ \theta' = \{3 : g \mapsto \hat{g}\} \\
\mathcal{S}[\![null?g]\!]\theta' & = & \langle \mathcal{S}[\![null?]\!]_v\emptyset\&sv_1, sf_1\rangle \\
 & & where\ \langle sv_1, sf_1\rangle = (\mathcal{S}_f[\![null?]\!]\emptyset)(\hat{g}\emptyset) \\
\mathcal{S}[\![null?]\!]\emptyset & = & \mathcal{K}_s[\![null?]\!] = \langle ABS, \lambda\hat{x}s.\langle \hat{x}s_v, serr\rangle\rangle \\
 & & \langle sv_1, sf_1\rangle = (\lambda\hat{x}s.\langle \hat{x}s_v, serr\rangle)(\hat{g}) = \langle \hat{g}_v, serr\rangle \\
\mathcal{S}[\![null?g]\!]\theta' & = & \langle ABS\&\hat{g}_v, serr\rangle = \langle \hat{g}_v, serr\rangle \\
\hat{f}\theta & = & \langle ABS, \lambda\hat{g}.\langle \hat{g}_v \ \&\ (ABS \sqcup \mathcal{S}_v[\![+1\ (f\ g\ (\lambda h_2.\lambda t_1.t)))]\!]\theta'), serr\rangle\rangle \\
 & & where\ \theta' = \{3 : g \mapsto \hat{g}\} \\
g_1 & = & g\ (\lambda h_2.\lambda t_1.t)) \\
\mathcal{S}[\![+1\ (fg_1)]\!]\theta' & = & \langle \mathcal{S}_v[\![+1]\!]\emptyset\&sv_1, sf_1\rangle \\
 & & where\ \langle sv_1, sf_1\rangle = (\mathcal{S}_f[\![+1]\!]\emptyset)(\mathcal{S}[\![fg_1]\!]\emptyset) \\
\mathcal{S}[\![+\ 1]\!] & = & \langle \mathcal{S}_v[\![+\ 1]\!]\&sv_2, sf_2\rangle \\
 & & where\ \langle sv_2, sf_2\rangle = (\mathcal{S}_f[\![+]\!])(\mathcal{S}[\![1]\!]) \\
\mathcal{S}[\![+]\!] & = & \mathcal{K}_s[\![+]\!] = \langle ABS, \lambda\hat{x}.\langle ABS, \lambda\hat{y}.\langle \hat{x}_v\&\hat{y}_v, serr\rangle\rangle\rangle \\
\mathcal{S}[\![1]\!] & = & \mathcal{K}_s[\![1]\!] = \langle ABS, serr\rangle \\
 & & \langle sv_2, sf_2\rangle = (\lambda\hat{x}.\langle ABS, \lambda\hat{y}.\langle \hat{x}_v\&\hat{y}_v, serr\rangle\rangle)(\mathcal{K}_s[\![1]\!]) \\
 & & \langle sv_2, sf_2\rangle = \langle ABS, \lambda\hat{y}.\langle \mathcal{K}_{sv}[\![1]\!]\&\hat{y}_v, serr\rangle\rangle \\
 & & \langle sv_2, sf_2\rangle = \langle ABS, \lambda\hat{y}.\langle ABS\&\hat{y}_v, serr\rangle\rangle \\
 & & \langle sv_2, sf_2\rangle = \langle ABS, \lambda\hat{y}.\langle \hat{y}_v, serr\rangle\rangle \\
\mathcal{S}[\![+1]\!] & = & \langle ABS\&ABS, \lambda\hat{y}.\langle \hat{y}_v, serr\rangle\rangle \\
\mathcal{S}[\![+1]\!] & = & \langle ABS, \lambda\hat{y}.\langle \hat{y}_v, serr\rangle\rangle \\
 & & \langle sv_1, sf_1\rangle = (\lambda\hat{y}.\langle \hat{y}_v, serr\rangle)(\mathcal{S}[\![fg_1]\!]) \\
 & & \langle sv_1, sf_1\rangle = (\langle \mathcal{S}_v[\![fg_1]\!], serr\rangle \\
\mathcal{S}[\![+1\ (fg_1)]\!]\theta' & = & \langle ABS\&\mathcal{S}_v[\![fg_1]\!], serr\rangle \\
\mathcal{S}[\![+1\ (fg_1)]\!]\theta' & = & \langle \mathcal{S}_v[\![fg_1]\!], serr\rangle \\
\hat{f}\theta & = & \langle ABS, \lambda\hat{g}.\langle \hat{g}_v \ \&\ (ABS \sqcup \mathcal{S}_v[\![f\ g_1]\!]), serr\rangle\rangle \\
\mathcal{S}[\![f\ g_1]\!] & = & \langle \mathcal{S}_v[\![f]\!]\&sv, sf\rangle \\
 & & where\ \langle sv, sf\rangle = (\mathcal{S}_f[\![f]\!])(\hat{g_1}) \\
\mathcal{S}[\![f\ g_1]\!] & = & \langle \hat{f}_v\&(\hat{f}_f\ \hat{g_1})_v\ ,\ (\hat{f}_f\ \hat{g_1})_f\rangle \\
\hat{f} & = & \langle ABS, \lambda\hat{g}.\langle \hat{g}_v\&(ABS \sqcup (\hat{f}_v\&(\hat{f}_f g_1)_v)), serr\rangle\rangle \\
 & & \\
\hat{g}_1 & = & \mathcal{S}[\![g_1]\!] = \mathcal{S}[\![g\ (\lambda h_2.\lambda t_1.t)]\!] = \langle \mathcal{S}_v[\![g]\!]\&sv, sf\rangle \\
 & & where\ \langle sv, sf\rangle = (\mathcal{S}_f[\![g]\!])(\mathcal{S}[\![\lambda h_2.\lambda t_1.t]\!]) \\
\hat{g}_1 & = & \langle \hat{g}_v\&(\hat{g}_f\ \hat{tt})_v, (\hat{g}_f\ \hat{tt})_f\rangle \quad where\ tt = \lambda h_2.\lambda t_1.t
\end{array}$$

$$
\begin{array}{rcl}
\mathcal{S}[\![\lambda h_2.\lambda t_1.t]\!]\emptyset & = & \langle ABS, \lambda\hat{h}.\ \mathcal{S}[\![\lambda t_1.t]\!]\delta'\rangle \\
 & & where\ \delta' = \emptyset \cup \{2 : h \mapsto \hat{h}\} \\
\mathcal{S}[\![\lambda t_1.t]\!]\delta' & = & \langle ABS, \lambda\hat{t}.\ \mathcal{S}[\![t]\!]\delta''\rangle \\
 & & where\ \delta'' = \delta' \cup \{1 : t \mapsto \hat{t}\} \\
\mathcal{S}[\![t]\!]\delta'' & = & \langle \delta''(\mathcal{S}[\![t]\!]), serr\rangle \\
 & = & \langle (\delta_2(\mathcal{S}[\![t]\!]), \delta_1(\mathcal{S}[\![t]\!])), serr\rangle \\
 & = & \langle (ABS, \hat{t}), serr\rangle \\
 & & \\
\mathcal{S}[\![\lambda h_2.\lambda t_1.t]\!] & = & \langle ABS, \lambda\hat{h}.\langle ABS, \lambda\hat{t}.\langle (ABS, \hat{t}), serr\rangle\rangle\rangle \\
\hat{tt} & = & \mathcal{S}[\![\lambda h_2.\lambda t_1.t]\!]) \\
g & = & \lambda c.c\ x_1\ g_1 \quad where\ x_1 = g\ hh\ and\ g_1 = g\ tt \\
 & & \\
\hat{g} & = & \langle ABS, \lambda\hat{c}.\mathcal{S}[\![c\ x_1\ g_1]\!]\rangle \\
\mathcal{S}[\![c\ x_1\ g_1]\!] & = & \langle \mathcal{S}_v[\![c\ x_1]\!]\&sv1, sf1\rangle \\
 & & where\ \langle sv_1, sf_1\rangle\ =\ (\mathcal{S}_f[\![c\ x_1]\!])(\hat{g_1}) \\
\mathcal{S}[\![c\ x_1]\!] & = & \langle \mathcal{S}[\![c]\!]\&sv_2\rangle \\
 & & where\ \langle sv_2, sf_2\rangle\ =\ (\mathcal{S}_f[\![c]\!])(\hat{x}_1) \\
\hat{g} & = & \langle ABS, \lambda\hat{c}.\langle \hat{c}_v\&(\hat{c}_f\hat{x}_1)_v\&((\hat{c}_f\hat{x}_1)_f\hat{g}_1)_v, ((\hat{c}_f\hat{x}_1)_f\hat{g}_1)_f\rangle\rangle \\
\hat{g}_f\hat{tt} & = & (\lambda\hat{c}.\langle \hat{c}_v\&(\hat{c}_f\hat{x}_1)_v\&((\hat{c}_f\hat{x}_1)_f\hat{g}_1)_v, ((\hat{c}_f\hat{x}_1)_f\hat{g}_1)_f\rangle)(\hat{tt}) \\
\hat{g}_f\hat{tt} & = & \langle \hat{tt}_v\&(\hat{tt}_f\hat{x}_1)_v\&((\hat{tt}_f\hat{x}_1)_f\hat{g}_1)_v, ((\hat{tt}_f\hat{x}_1)_f\hat{g}_1)_f\rangle \\
\hat{g}_f\hat{tt} & = & \langle ABS\&(ABS)\&((\lambda\hat{t}.\langle (ABS, \hat{t}_v), serr\rangle)\hat{g}_1)_v, ((\lambda\hat{t}.\langle (ABS, \hat{t}_v), serr\rangle)\hat{g}_1)_f\rangle \\
\hat{g}_f\ \hat{tt} & = & \langle (ABS, (\hat{g}_1)_v), serr\rangle \\
\hat{g}_1 & = & \langle \hat{g}_v\&(ABS, (\hat{g}_1)_v), serr\rangle \\
 & & \\
\hat{f} & = & \langle ABS, \lambda\hat{g}.\langle \hat{g}_v\&(ABS \sqcup (\hat{f}_v\&(\hat{f}_f\hat{g}_1)_v)), serr\rangle\rangle \\
where\ \hat{g}_1 & = & \langle \hat{g}_v\&(ABS, (\hat{g}_1)_v), serr\rangle
\end{array}
$$

We have to compute the least fixpoint of $\hat{f}$ by constructing Kleene's ascending chain of approximations. We have to compute also the least fixpoint of $\hat{g}_1$ and the AKC construction of approximations starts with $\langle (ABS, STR), serr\rangle$.

$$
\begin{array}{rcl}
\hat{g}_1^{(0)} & = & \langle (ABS, STR), serr\rangle \\
\hat{g}_1^{(1)} & = & \langle \hat{g}_v\&(ABS, (ABS, STR)), serr\rangle \\
\hat{g}_1^{(1)} & = & \langle \hat{g}_v\&(ABS, STR), serr\rangle \\
 & & \\
\hat{g}_1^{(2)} & = & \langle \hat{g}_v\&(ABS, (\hat{g}_v\&(ABS, STR))), serr\rangle \\
\hat{g}_1^{(2)} & = & \langle \hat{g}_v\&(ABS, (\hat{g}_v\&ABS, \hat{g}_v\&STR)), serr\rangle \\
\hat{g}_1^{(2)} & = & \langle \hat{g}_v\&(\hat{g}_v\&ABS, \hat{g}_v\&STR), serr\rangle \\
\hat{g}_1^{(2)} & = & \langle \hat{g}_v\&(\hat{g}_v)\&(ABS, STR), serr\rangle \\
\hat{g}_1^{(2)} & = & \langle \hat{g}_v\&(ABS, STR), serr\rangle \\
 & \cdots & \\
\hat{g}_1 & = & \langle \hat{g}_v\&(ABS, STR), serr\rangle
\end{array}
$$

$$
\begin{aligned}
\hat{f}^{(0)} &= \langle FAIL, serr\rangle \\
\hat{f}^{(1)} &= \langle ABS, \lambda\hat{g}.\langle \hat{g}_v \& (ABS \sqcup (FAIL \& (\hat{f}_f \hat{g}_1)_v)), serr\rangle\rangle \\
\hat{f}^{(1)} &= \langle ABS, \lambda\hat{g}.\langle \hat{g}_v \& (ABS \sqcup FAIL), serr\rangle\rangle \\
\hat{f}^{(1)} &= \langle ABS, \lambda\hat{g}.\langle \hat{g}_v \& ABS, serr\rangle\rangle \\
\hat{f}^{(1)} &= \langle ABS, \lambda\hat{g}.\langle \hat{g}_v, serr\rangle\rangle \\
\\
\hat{f}^{(2)} &= \langle ABS, \lambda\hat{g}.\langle \hat{g}_v \& (ABS \sqcup (ABS \& ((\lambda\hat{g}.\langle \hat{g}_v, serr\rangle)\hat{g}_1)_v)), serr\rangle\rangle \\
\hat{f}^{(2)} &= \langle ABS, \lambda\hat{g}.\langle \hat{g}_v \& (ABS \sqcup (\hat{g}_1)_v), serr\rangle\rangle \\
\hat{f}^{(2)} &= \langle ABS, \lambda\hat{g}.\langle \hat{g}_v \& (\hat{g}_1)_v, serr\rangle\rangle \\
\hat{f}^{(2)} &= \langle ABS, \lambda\hat{g}.\langle \hat{g}_v \& (\hat{g}_v \& (ABS, STR)), serr\rangle\rangle \\
\hat{f}^{(2)} &= \langle ABS, \lambda\hat{g}.\langle \hat{g}_v \& (ABS, STR), serr\rangle\rangle \\
\\
\hat{f}^{(3)} &= \langle ABS, \lambda\hat{g}.\langle \hat{g}_v \& (ABS \sqcup (ABS \& ((\lambda\hat{g}.\langle \hat{g}_v \& (ABS, STR), serr\rangle)\hat{g}_1)_v)), serr\rangle\rangle \\
\hat{f}^{(3)} &= \langle ABS, \lambda\hat{g}.\langle \hat{g}_v \& (ABS \sqcup ((\hat{g}_1)_v \& (ABS, STR))), serr\rangle\rangle \\
\hat{f}^{(3)} &= \langle ABS, \lambda\hat{g}.\langle \hat{g}_v \& (\hat{g}_1)_v \& (ABS, STR), serr\rangle\rangle \\
\hat{f}^{(3)} &= \langle ABS, \lambda\hat{g}.\langle \hat{g}_v \& (\hat{g}_v \& (ABS, STR)) \& (ABS, STR), serr\rangle\rangle \\
\hat{f}^{(3)} &= \langle ABS, \lambda\hat{g}.\langle \hat{g}_v \& (ABS \& ABS, STR \& STR), serr\rangle\rangle \\
\hat{f}^{(3)} &= \langle ABS, \lambda\hat{g}.\langle \hat{g}_v \& (ABS, STR), serr\rangle\rangle \\
&\cdots \\
\hat{f} &= \langle ABS, \lambda\hat{g}.\langle \hat{g}_v \& (ABS, STR), serr\rangle\rangle
\end{aligned}
$$

So f is tail-strict.

5 Conclusion

This work has provided a method to deal with higher-order strictness analysis over non-flat domains. This method is general enough to deal with the "n-th" strictness of a component of any kind of data structures.

It could be used as a basic framework to two further extensions : 1) strictness analysis in the context of an entire program (by the means of a collecting interpretation of expressions [6]) and 2) polymorphic strictness analysis (using the notion of category theory [10]).

6 Acknowledgements

This work owes a debt to the Functional Programming Group at Yale University, New Haven. I would particularly like to thank Paul Hudak, who suggested this subject and gave me much encouragement and a lot of his time. Thanks also to the members of the team for all their support. Thanks also to Michel Mauny, INRIA Rocquencourt, for the final comments of this report.

References

[1] S. Abramsky. *Strictness Analysis and Polymorphic Invariance.* Proceedings of the DIKU Workshop on Programs as Data Objects, LNCS 217, Springer-Verlag, 1986.

[2] G.L. Burn. *Evaluation Transformers - A Model for the Parallel Evaluation of Functional Languages.* In Proceedings of 1987 Functional Programming Languages and Computer Architecture Conference, pages 446-470, Springer Verlag LNCS 274, September 1987.

[3] G.L. Burn, C.L. Hankin, and S. Abramsky. *The theory of strictness analysis for higher-order functions.* In LCNS 217: Programs as Data Objects, pages 42-62, Springer-Verlag, 1985.

[4] P. Cousot and R. Cousot. *Abstract interpretation : a unified lattice model for static analysis of programs by construction or approximation of fixpoints.* In 4th ACM Symposium on Principles of Programming Languages, pages 238-252,ACM,1977.

[5] P. Hudak and J. Young. *Higher-order strictness analysis for untyped lambda calculus.* In 12th ACM Symposium on Principles of Programming Languages, pages 97-109, January 1986.

[6] P. Hudak and J. Young. *A collecting interpretation of expressions (without powerdomains).* In 15th ACM Symposium on Principles of Programming Languages, pages 107-118, January 1988.

[7] P. Hudak and P. Wadler (editors). *Report on the Functional Programming Language Haskell.* Technical Report YALEU/DCS/RR-666, Yale University, Department of Computer Science, December 1988.

[8] A. Mycroft. *Abstract interpretation and optimizing transformations for applicative programs.* PhD thesis, University of Edinburgh,1981.

[9] S.L. Peyton-Jones. *The Implemention of Functional Programming Languages.* Prentice Hall, 1987.

[10] B. C. Pierce. *A Taste of Category Theory for Computer Scientists.* Technical Report CMU-CS-88-203, 1988.

[11] D.A. Schmidt. *Denotational Semantics - A Methodology for Languages Development.* Allyn and Bacon, Inc., Boston, Mass.,1986.

[12] J.E. Stoy. *The Scott-Strachey Approach to Programming Language Theory.* The M.I.T. Press, Mass, 1977.

[13] P.Wadler. *Strictness analysis on non-flat domains (by abstract interpretation over finite domains).* In S. Abramsky and C. Hankin, editors, "Abstract Interpretation of Declaratives Languages", Ellis Horwood, 1987.

[14] P.Wadler and R.J.M. Hughes. *Projections for strictness analysis.* In Proceedings of 1987 Functional Programming Languages and Computer Architecture Conference, pages 385-407, Springer Verlag LNCS 274, september 1987.

Imprimé en France
par
l'Institut National de Recherche en Informatique et en Automatique